Environment Diagrams
1. Why do set!, when we can always just redefine a variable using a define statement? Instead of doing (set! x 3), why don’t we just do (define x 3) again? I propose the following alternative implementation of counter.

The Old Way

(define count
  (let ((current 0))
    (lambda()
      (set! current (+ 1 current))
      current)))
(count) ==> 1
(count) ==> 2

My Brilliant New Way

(define count
  (let ((current 0))
    (lambda ()
      (define current
        (+ current 1))
      current)))

How dumb am I? What happens when I use my new brilliant implementation?

2. Define a procedure f so that, given the procedure call (+ (f 0) (f 1)) if STk evaluates from left to right, it returns 0, and if STk evaluates from right to left, it returns 1.

3. Define a procedure fib so that, every time it is called, it returns the next Fibonacci number, starting from 1:
(fib) ==> 1; (fib) ==> 2; (fib) ==> 3; (fib) ==> 5; (fib) ==> 8, etc.
4. Draw environment diagrams for each of the following:

```
(define (f + -) (+ ((lambda(-) (- 3 5)) -) 10))
(f - +)

(define (hmm n) (lambda(x) (+ x y n)))
(define (uhh y)
  (define hmm-y (hmm y))
  (hmm-y 2))
(uhh 42)

(define answer 0)
(define (square f x)
  (let ((answer 0))
    (f x)
    answer))
(square (lambda(n) (set! answer (* n n))) 3)
```
(define a 3)
((lambda(a)
  ((lambda(a) (a))
   (lambda() (set! a 'myxomatosis)))
  a)
 (* a a))

(define a 'scatterbrain)
((lambda(a b) (b) a)
  a
  (let ((b 'cuttooth))
    (lambda() (set! a b)))
  a)
(define slow-op-maker op)
  (let ((old-result #f))
    (lambda(x)
      (let ((return old-result))
        (set! old-result (op x))
        return))))

(define slow-sqr (slow-op-maker square))
(slow-sqr 3)
(slow-sqr 5)
(define slow-cube (slow-op-maker cube))
(slow-cube 3)
(slow-cube 5)